ARO 22912.8-EL

(2)

AUTOMATIC COMPILER CONSTRUCTION

FINAL REPORT

VINCENT P. HEURING
WILLIAM M. WAITE
GERHARD FISCHER

FEBRUARY 1, 1990

U.S. ARMY RESEARCH OFFICE

CONTRACT/DAAL 03-86-K-0100

UNIVERSITY OF COLORADO AT BOULDER
ELECTRICAL AND COMPUTER ENGINEERING
DEPARTMENT

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): 153-6923	5. MONITORING ORGANIZATION REPORT NUMBER(S): ARO 22912.8-EL
6a. NAME OF PERFORMING ORGANIZATION / 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: U. S. Army Research Office
6c. ADDRESS (City, State, and ZIP Code): University of Colorado, ECE Department, Campus Box 425, Boulder, Colorado 80309-0425	7b. ADDRESS (City, State, and ZIP Code): P. O. Box 12211, Research Triangle Park, NC 27709-2211
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: U. S. Army Research Office / 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: DAAL03-86-K-0100
8c. ADDRESS (City, State, and ZIP Code): P. O. Box 12211, Research Triangle Park, NC 27709-2211	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): Automatic Compiler Construction

12. PERSONAL AUTHOR(S): Vincent P. Heuring, William M. Waite

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 6/15/86 TO 6/14/89	February 2, 1990	6

16. SUPPLEMENTARY NOTATION: The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Compiler Construction Expert Systems Automatic Programming

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The system that we have developed with ARO support during the past three years convincingly demonstrates the technology required to:
- manage complex user requests within the context of a large tool suite.
- allow users to obtain instruction about how to make complex requests from the system itself and
- easily develop processors for problem-oriented specification languages.

We have used the system routinely in its own development and in a variety of applications from circuit board layout through robot control to database query specification. It is now available for distribution to sites with Sun3 and Sun4 computers.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

The system that we have developed with ARO support during the past three years convincingly demonstrates the technology required to

- manage complex user requests within the context of a large tool suite,
- allow users to obtain instruction about how to make complex requests from the system itself, and
- easily develop processors for problem-oriented specification languages.

We have used the system routinely in its own development and in a variety of applications from circuit board layout through robot control to database query specification. It is now available for distribution to sites with Sun3 and Sun4 computers.

This report summarizes the most important results of our research, and places them in the context of the general problem of transitioning the technology represented by large software systems. The work described here relates directly to the growing problem of moving results out of the laboratory into day-to-day life. We show how the inevitable complexity of a modular collection of processes can be hidden from the user by an expert system that understands their low-level relationships. That expert system can observe the behavior of the hidden processes and interpret their actions in terms the user understands. It can also call on extensive tutorial material, including exercises for the user, to aid the user in completing a task. In order to demonstrate these principles, we have used them to construct a system that solves a real-life problem — construction of processors for problem-oriented specification languages.

1. The Problem — Using Computers More Effectively

Any computer, from a small PC to a large mainframe, provides an environment within which a user makes requests. The requests may simply extract information from the environment (e.g. a request to obtain the size of a file) or they may alter the environment (e.g. a request to compile a program and store the object file). Some requests are simple to implement and others are more complex, but each appears to the user as an atomic operation that carries out some useful task. One way to use the computer more effectively is to "package" commonly-used combinations of requests as single requests. Most operating systems provide command script mechanisms for this purpose, and special tools like the Unix make[1] may also be available. Unfortunately, neither of these approaches is entirely satisfactory. Command scripts are too rigid, and make-like tools neither provide sufficient parameterization nor sufficient hiding of intermediate products.

Many requests made of computers require large-scale parameterization: The class of problems to be solved by the request is well-understood, but there are a large number of problem instances that must be solved in slightly different ways. A typical example of such a situation is a request to sort some data file. The general sorting problem is well-understood, but a particular solution depends upon many things (collating sequence, primary and secondary fields, characters to be ignored, etc.) The general approach to such problems is to create processor generators that accept a description of the problem instance and create a processor to handle requests for its solution. Problem descriptions are written in declarative notations that have often been called *fourth-generation languages*.

Creation of a processor generator is basically a compiler construction task. The fourth-generation problem-description language must be designed, and a program built to analyze descriptions in this language and produce processors that solve the problems described. Requiring that all processor generators be built by people trained in compiler construction would, however, seriously limit the availability of this problem solving technique. It is much better to provide people who need processor generators with the ability to construct them directly.

The goal of our research project was a system that would allow a person needing a processor generator to write declarative specifications describing the desired fourth-generation language and its translation, and then request construction of the specified processor generator. In order to produce such a system, we had to manage the request to produce the processor generator from its specification (which is very complex, but must appear atomic to the user). In order to make the system usable to people other than its designers, we had to provide easy ways for users to learn to use it and to obtain aid in interpreting diagnostic output.

2. Management of Complex User Requests

Our first problem was how to manage a request for processor generation. We knew that such a request would involve activation of a number of separate tools to create processor components and to integrate those components into a functioning whole. Many of the individual tools would be obtained from other sources, because the cost of redeveloping them in-house would be prohibitive. That meant that whatever management mechanism we chose could not require specific tool interfaces. Since the users of the system would not be experts in compiler construction it was important to hide both the tools and the intermediate products. Although we needed to allow the user to create simple processors simply, facilities also needed to be available for creation of complex processors that did not precisely fit a standard model. Thus the creation request needed to be parameterizable. Finally, the management tool needed to be flexible so that we could add new tools and change existing tools on the basis of our experience during the project.

We chose Odin,[2,3] an expert system whose domain of expertise is management of complex user requests, as our management tool.[4] Odin, like any expert system, separates the *inference engine* that manages the user's request from the *knowledge base* that specifies the management policies. The manufacture of a processor is described by a *derivation graph*,[5] a declarative specification from which Odin's knowledge base can be generated. Because the derivation graph is a declarative specification rather than a program, it is easy to change and can be checked for consistency. This allows us the required flexibility to add and alter tools. Odin's inference engine is completely independent of the processor-generation problem, and could be used to manage any collection of software tools.

Nodes in the derivation graph represent either tools or objects. Tools are actually implemented by operating system command scripts. Odin allows the node to choose the command processor it will use, the directory in which it will run, and the commands it will execute. Thus the actions taken by a node can be tailored to the tool it invokes, permitting us to employ arbitrary tools for which only object code is available, or which require input or output format changes. This allows use of off-the-shelf tools and avoids having to do extensive tool development in-house. Again, the behavior is independent of the processor-generation problem; *any* collection of arbitrary programs could be managed.

The derivation graph is normally invisible to the user, although it is possible to get explanations of what Odin is doing in terms of the derivation graph. (A hallmark of an expert system is that it can explain the reasoning by which it arrives at a requested result.) This means that we can increase the complexity of the manufacturing process without increasing the cognitive load on the user. Intermediate products used during the manufacture are also invisible. They are kept in a separate directory called the *cache*. Wherever possible, Odin will use existing copies of intermediate object in satisfying requests, thus reducing manufacturing costs.

During the grant period, we have worked closely with the developer of Odin in devising techniques for derivation graph specification that increase flexibility and simplify processing. We now have standard approaches to the introduction of new tools, and very powerful mechanisms for varying the manufacturing steps on the basis of the particular set of specifications supplied by the user. This latter improvement has drastically reduced the need for user parameters to control manufacture.

3. Error Reporting and Documentation

Error reporting is a critical problem when dealing with complex user requests. The situation is analogous to that found in a complex program when a low-level component detects an error, but does not have sufficient context to report that error in terms understandable to the user. The technique of "unhurried diagnostics" was developed to deal with this problem.[6] A program having information about a failure may take any one of four actions:

1) Output the information and terminate.

2) Output the information and proceed by an alternate route.

3) Suppress the information and proceed by an alternate route.

4) If not the main program, pass the information to its caller and indicate failure.

The original version of Odin allowed each of the first three actions to be taken by individual nodes; the fourth was added during the grant period.

The important point is that the derivation graph can specify arbitrary manufacturing steps to be applied to error reports. Those manufacturing steps can access arbitrary information about both the derivation graph and the current contents of the cache. Thus it is possible to apply a complex interpretation process to reports provided by any tool, without altering the tool in any way. A user of the system is unaware of this interpretation process, and need know nothing about the tool issuing the report.

We have increased the amount of information available for error reporting by placing the system documentation on line in hypertext form. Not only does this allow the user to browse the documentation in order to answer question arising while creating specifications, but it permits the error analysis to access the documentation needed to explain a failure. There is no need for the user to have the printed form of the system documentation at hand because a simple request will place them in hypertext browsing mode at the most likely explanation of their problem. If the system is wrong, the user is in a position to browse the entire set of documentation if necessary.

Both the hypertext form and the printed form of the system documentation are generated from a single body of text. Thus we avoid any possible inconsistencies and reduce the cost of producing the hypertext to zero.

We have made a modification to the hypertext browser that allows a reader to modify and execute examples given in the documentation. This feature is the basis of a system tutorial for self-paced learning. A set of graded examples with accompanying exercises introduces the new user to the individual specification methods supported by the system. Our experience has been that this approach to teaching people how to use the system is much more effective than either a printed manual alone or a sequence of lectures.

4. Processor Generation

We began this research project with the usual complement of compiler construction tools: a lexical analyzer generator,[7] a parser generator[8] and an attribute grammar processor.[9] After building a derivation graph that managed these tools and a collection of scripts to make them work together we had a rudimentary system for generating language processors. There were two basic deficiencies in this system that we have attacked during the grant period — the weakness of the specification techniques and the poor performance of the generated processors.

If a language processor is specified by a set of regular expressions (input to the lexical analyzer generator), a context-free grammar (input to the parser generator) and an attribute grammar then most of the processor's behavior must be specified algorithmically within the attribute grammar. Our research has identified several additional subproblems that can be specified declaratively instead of algorithmically. We have used the system to generate processors for fourth-generation languages in which these declarative specifications can be written. By extending the derivation graph, we incorporated these processors into the system. A user does not need to know of the existence of the additional processors. If specifications in the languages they implement are provided to the system, they are invoked; otherwise they are not.

A parallel effort involves the individual tools themselves. In response to experience with our lexical analyzer generator we redefined the specification language to increase its flexibility and to allow for library specifications. We replaced the parser generator with a new version from Germany, increasing the speed of the generated processor by a factor of five without altering the specification language. The system now supports two attribute grammar analyzers, selecting one or the other on the basis of the type of specification file provided by the user. Extensive measurements of generated processors and comparisons with hand coded versions have led to modifications of the system and better results.

One of the more difficult tasks when specifying a language processor seems to be the design of the context-free grammar that describes program structure. In order to construct an efficient parser for the language, the grammar must satisfy certain constraints. When these constraints are not satisfied, it is often difficult for the designer to determine how the grammar should be changed. The constraint violation is sometimes the result of a subtle ambiguity in the language definition, and sometimes simply due to a blunder in writing the grammar. We have developed a processor that is capable of pinpointing the most common ambiguities and fixing up the blunders. This processor dramatically reduces the difficulty of specifying grammars.

5. Conclusion

Our original proposal was to apply Artificial Intelligence techniques to the problem of automatic compiler construction. We said that the work would result in an integrated development environment that provides:

- greatly reduced compiler development time
- compilers as fast and compact as those constructed by hand
- guaranteed compiler reliability
- a reduction of human expertise required in compiler construction
- a simple path for compiler maintainability
- greatly reduced life-cycle cost.

These objectives have been met. We are currently distributing the system, called Eli, for Sun3 and Sun4 equipment. Our experience with a wide variety of language processors indicates that development time is about 1/3 of that required using other approaches. Our measurements of a generated C compiler indicate that it has approximately the same performance as gcc, a well-regarded compiler that was coded by hand. Compilers generated by Eli do not crash. They always behave exactly according to the specifications from which they were generated. We have used Eli in a project class at the University of Colorado for three years, with students who have no prior compiler construction experience. They are able to create relatively sophisticated language processors without becoming compiler construction experts. Since the processors are generated from specifications, compiler maintenance is reduced to maintenance of the specifications. Many parts of these specifications are re-usable, thus reducing the total life-cycle cost of each compiler.

In addition to providing the specific development environment for language processors, we have demonstrated the technology needed to construct such environments in general. This technology can be applied to any situation in which a number of off-the-shelf software components must be combined to satisfy a single complex request. Our approach can be used to create a flexible system that provides error reports at an appropriate level, extensive documentation and help facilities, and is capable of tutoring its users.

6. References

1. S. I. Feldman, 'Make — A Program for Maintaining Computer Programs', *Software—Practice & Experience*, **9**, (April 1979).
2. G. M. Clemm, 'The Odin System - An Object Manager for Software Environments', Ph.D. Thesis, Department of Computer Science, University of Colorado, Boulder, CO, 1986.
3. G. M. Clemm, 'The Odin Specification Language', in *International Workshop on Software Version and Configuration Control '88*, Teubner, Stuttgart, FRG, 1988.
4. W. M. Waite, V. P. Heuring and U. Kastens, 'Configuration Control in Compiler Construction', in *Proceedings of the International Workshop on Software Version and Configuration Control*, Teubner, Stuttgart, FRG, 1988.
5. E. Borison, 'A Model of Software Manufacture', in *Advanced Programming Environments*, vol. 244, R. Conradi, T. M. Didriksen and D. H. Wanvik, (eds.), Springer Verlag, Heidelberg, 1986.
6. W. S. Brown, 'An Operating Environment for Dynamic-Recursive Computer Programming Systems', *Communications of the ACM*, **8**, 371-377 (June 1965).
7. V. P. Heuring, 'The Automatic Generation of Fast Lexical Analyzers', *Software—Practice & Experience*, **16**, 801-808 (September 1986).
8. P. Dencker, K. Dürre and J. Heuft, 'Optimization of Parser Tables for Portable Compilers', *ACM Transactions on Programming Languages and Systems*, **6**, 546-572 (October 1984).
9. U. Kastens, B. Hutt and E. Zimmermann, *GAG: A Practical Compiler Generator*, Springer Verlag, Heidelberg, 1982.